ELSEVIER

Theoretical Computer Science 173 (1997) 209–233

Theoretical Computer Science

# A confluent calculus for concurrent constraint programming

Kim Marriott [a], Martin Odersky [b,*]

[a] *Monash University, Clayton 3168, Victoria, Australia*

[b] *Institut fur Programmstrukturen und Datenorganisation, Universität Karlsruhe, 76128 Karlsruhe, Germany*

**Abstract**

Confluence is an important and desirable property as it allows the program to be understood by considering any desired scheduling rule, rather than having to consider all possible schedulings. Unfortunately, the usual operational semantics for concurrent constraint programs is not confluent as different process schedulings give rise to different sets of possible outcomes. We show that it is possible to give a natural confluent calculus for concurrent constraint programs, if the syntactic domain is extended by a blind choice operator and a special constant standing for a discarded branch. This has application to program analysis.

## 1. Introduction

Concurrent constraint programming (ccp) [20, 19] is a recent paradigm which elegantly combines logical concepts and concurrency mechanisms. The computational model of ccp is based on the notion of a *constraint system*, which consists of a set of constraints and an *entailment* relation. Processes interact through a common *store*. Communication is achieved by *telling* (adding) a given constraint to the store, and by *asking* (checking whether the store entails) a given constraint. Standard ccp provides a non-deterministic guarded choice operator. In the operational semantics of ccp, non-determinism arises in two different ways. First, if the guards of two branches in a committed choice construct are both entailed by the store either branch can be picked. Second, different process schedulings (that is, interleavings of transitions) can lead to different results since a given process scheduling can prune the decision space by selecting a branch in a committed choice before strengthening the store. In this way, some branches that would be entailed by the stronger store might be excluded by the weaker one. This second source of non-determinism means that to find the possible outcomes of a program all process schedulings must be considered in the operational

* Corresponding author. E-mail: odersky@ira.uka.de.


*PII* S0304-3975(96)00196-X

semantics. This need to consider all process schedulings also holds for the denotational semantics of ccp, which expresses parallel composition by interleaving.

Because of the combinatorial explosion of reduction sequences, an interleaving semantics makes reasoning about possible evaluations cumbersome. Yet such reasoning is necessary for many tasks in program analysis, verification and transformation. This contrasts to the situation in both the lambda calculus and (idealised) Prolog. The semantics for both have confluence properties that make it unnecessary to consider different-process schedulings. In the lambda calculus, confluence is embodied in the Church–Rosser theorem [1], which says that different reduction sequences starting from the same term can always be rejoined in a common reduct. As a consequence, evaluation in the lambda calculus is deterministic. In Prolog, confluence is embodied in the Switching Lemma [11], which ensures that different literal selection strategies give rise to the same set of answers.

In the context of concurrency, confluence is an even more desirable property since concurrent programs are notoriously difficult to reason about and to analyse. Unfortunately, as we have seen, despite monotonicity of communication, the standard operational semantics for ccp languages is not confluent in the sense that different process schedulings can give rise to different outcomes. This is because of the guarded choice. Indeed, it has become part of the programming language folklore that it is impossible to have both guarded choice and confluence.

We present here a calculus for ccp that is equivalent to ccp's standard semantics in that both lead to the same observations, yet is confluent. Actually, we give a calculus for a slightly larger language, $\mathsf{ccp}_{+0}$, which extends ccp by providing a *blind choice* construct and a *failure* constant **0**. The main difference between our calculus for $\mathsf{ccp}_{+0}$ and the standard operational semantics for ccp lies in the treatment of guarded choice. In ccp, once a choice is made, all other alternatives of a choice construct are discarded. In $\mathsf{ccp}_{+0}$, the other alternatives are kept around, but extended with a guarded branch which reduces to **0** on termination. This allows other alternatives to be considered during evaluation, but if they are still suspended when evaluation terminates, they are discarded. The calculus distinguishes between the two forms of non-determinism in ccp. Non-determinism arising from multiple guards being enabled is expressed by the blind choice operator in the term language. Process scheduling non-determinism is reflected by a choice among different reduction sequences, analogous to the situation in the lambda calculus. Our main result is a confluence theorem for this calculus, which essentially says that the choice of process scheduling has no influence on the observable behaviour. This is equivalent to the Church–Rosser theorem for the lambda calculus or the Switching Lemma for Prolog. Our result thus refutes the folklore that it is impossible to have both guarded choice and confluence. Monotonicity of communication is crucial to our result.

Besides its theoretical interest, our confluent calculus has at least two applications. The first application is to the static analysis of ccp. Lack of confluence in the usual operational semantics and denotational semantics means that program analysis cannot be directly based on these semantics, as the cost of considering all

process schedulings in an analysis is prohibitive. For this reason an approach to the analysis of ccp programs has been to base analyses on a non-standard operational semantics for ccp which is confluent but which approximates the usual ccp operational semantics by allowing more reductions [2, 3, 21, 7]. Analyses are then proved correct with respect to this approximate operational semantics. The disadvantage of this approach is an inherent loss of precision in the analysis because of the approximation introduced in the new semantics or in the program transformation. Our calculus, therefore, provides a better basis for analysis for two reasons. First, because the calculus is confluent, there is no need to introduce complex artificial semantics or transformations as efficient analysis can be directly based on the calculus. Second, because the calculus gives the same observational behaviour as the usual operational semantics, there is no inherent loss of precision and the analysis can be more accurate.

A second application of the confluent semantics is as the basis of a denotational semantics for ccp languages based on sets of closure operators. Saraswat et al. [19] gave a semantics for deterministic ccp agents in which the denotation was a closure operator. This was modified by Marriott et al. [13] to give a denotational semantics for constraint logic programming languages with delay (essentially these are ccp languages with blind choice but not guarded choice) by identifying the denotation of a subagent as a set of closure operators. In essence, the calculus we give here transforms guarded choice into blind choice by flagging some of the blind choice alternatives. This suggests that the semantic equations of [13] can be modified to give a denotational semantics for ccp with guarded choice by attaching a flag to the resting points of a closure operator indicating if that resting point is valid. Indeed, such a semantics is closely related to that recently given by Nyström [17] for ccp languages in which the denotation of a subagent is a function which maps an oracle, which is a sequence of non-deterministic choices, to a closure operator and a set of conditions which describe when it is legal to choose this branch.

Our result showing that the $\mathrm{ccp}_{+0}$ programs are confluent generalizes confluence results of Maher [12] and Saraswat et al. [19] about deterministic ccp subsets and Falaschi et al. [7] on the identification of subclasses of ccp for which the usual operational semantics is confluent. Montanari et al. [14] give a confluent operational semantics for a variant of ccp with both indeterminism (blind choice) and nondeterminism (angelic choice); however they do not consider guarded choice. Niehren and Smolka have introduced the $\delta$ [15] and $\rho$ [16] calculi which have strong connections to the $\pi$-calculus and deterministic ccp, respectively. They have shown that both of these calculi are confluent. However, unlike our calculus neither the $\rho$ nor the $\delta$ calculus has a non-deterministic guarded choice operator. Since the earlier version of this paper appeared, Nyström [17] has given a confluent denotational semantics for ccp programs based on oracles and closure operators.

The rest of this paper is organized as follows. Section 2 introduces the standard operational semantics of the ccp languages. Section 3 presents our calculus. Section 4 shows that reduction in our calculus is confluent and Section 5 shows that the

calculus and operational semantics of ccp are observationally equivalent. Section 6 sketches an application of our calculus to the analysis of ccp programs. Section 7 concludes.

## 2. Concurrent constraint programming

Concurrent constraint programming was proposed by Saraswat [20, 19]. We follow here the definition given in [19], which is based on the notion of cylindric constraint system.

A *cylindric constraint system* [8] is a structure $\mathbf{C} = \langle \mathscr{C}, \leqslant, \sqcup, \mathit{true}, \mathit{false}, \exists \rangle$ such that:

1. $\langle \mathscr{C}, \leqslant \rangle$ is a complete algebraic lattice, where $\sqcup$ is the lub operation (representing logical and), and *true*, *false* are the least and the greatest elements of $\mathscr{C}$, respectively;
2. For each $x \in \mathit{Vars}$ the function $\exists_x : \mathscr{C} \to \mathscr{C}$ is a *cylindrification operator*:
   (E1) $\exists_x c \leqslant c$,
   (E2) $c \leqslant c'$ implies $\exists_x c \leqslant \exists_x c'$,
   (E3) $\exists_x (c \sqcup \exists_x c') = \exists_x c \sqcup \exists_x c'$,
   (E4) $\exists_x \exists_y c = \exists_y \exists_x c$;
3. For each $x, y \in \mathit{Vars}$, $\mathscr{C}$ contains the *diagonal element*, $d_{xy}$, which satisfies
   (D1) $d_{xx} \leqslant \mathit{true}$,
   (D2) if $z \neq x, y$ then $d_{xy} = \exists_x (d_{xz} \sqcup d_{zy})$,
   (D3) if $x \neq y$ then $c \leqslant d_{xy} \sqcup \exists_x (c \sqcup d_{xy})$.

As usual, we take $c = c'$ iff $c \leqslant c' \wedge c' \leqslant c$. The cylindrification operators essentially model existential quantification and so are useful for defining a hiding operator in the language. Note that if $\mathbf{C}$ models the equality theory, then the diagonal element $d_{xy}$ can be thought of as the formula $x - y$.

Deviating slightly from the treatment of [19], we will base our exposition of ccp on renamings instead of diagonal elements. Renamings can be defined in terms of diagonal elements as follows.

**Definition.** Let $x$ and $y$ be variables and let $c \in \mathbf{C}$. Then the *renaming* $[y/x]c$ of $y$ for $x$ in $c$ is the constraint $\exists_x (d_{xy} \sqcup c)$.

**Definition.** The *free variables* $\mathrm{fv}(c)$ of $c \in \mathbf{C}$ is the set $\{x \mid \exists_x c \neq c\}$.

The following proposition shows that we can consistently rename the free variables of a constraint.

**Proposition 2.1.** *Let $c \in \mathbf{C}$ and let $x$ and $y$ be variables such that $y \notin \mathrm{fv}(c)$. Then* $\exists_y [y/x]c = \exists_x c$.

**Proof.** It follows from [8, Theorem 1.3.2] that $\exists_x d_{xy} = true$. We then compute as follows.

$$\begin{aligned}
\exists_y [y/x]c &= \exists_y \exists_x (d_{xy} \sqcup c) && \text{by definition of } [y/x] \\
&= \exists_x \exists_y (d_{xy} \sqcup c) && \text{by (E4)} \\
&= \exists_x \exists_y (d_{xy} \sqcup \exists_y c) && \text{since } y \notin \mathrm{fv}(c) \\
&= \exists_x (\exists_y d_{xy} \sqcup \exists_y c) && \text{by (E3)} \\
&= \exists_x \exists_y c && \text{since } \exists_y d_{xy} = true \\
&= \exists_x c && \text{since } y \notin \mathrm{fv}(c). \quad \square
\end{aligned}$$

The description and semantics of the **ccp** class of languages is parametric with respect to an underlying cylindric constraint system **C**. The syntax of agents $M$ and programs $P$ is given by the grammar:

$$\begin{aligned}
&\text{(Agent)} && M ::= c \mid R \mid p\bar{y} \mid M \cdot M \mid \exists_x M \\
&\text{(Choice)} && R ::= R \,[\!]\, R \mid c \mapsto M \\
&\text{(Program)} && P ::= D ; M \\
&\text{(Declarations)} && D ::= D, D \mid p\bar{x} := M
\end{aligned}$$

Two fundamental agents are the *tell* operation $c$ which adds the constraint $c$ to the store and the guarded choice among *ask* operations $[\!]_{i=1}^{n} c_i \mapsto M_i$ which evaluates some $M_i$, provided the corresponding *guard* $c_i$ is entailed by the store. An agent can also be a *procedure call* $p\bar{y}$, where $\bar{y}$ is a vector of parameters $(y_1, \ldots, y_m)$. We assume that every procedure identifier $p$ has exactly one declaration of the form $p(x_1, \ldots, x_m) := M$ in a program and that the lengths of actual and formal argument lists match. Agents can be combined using parallel composition $(\cdot)$. The quantifier $\exists_x M$ hides the use of variable $x$ inside the agent $M$. We will often use the word *term* as a synonym for *agent*.

*Free variables* $\mathrm{fv}(M)$ and *renamings* $[x/y]M$ have their usual inductive definitions, where the cases where $M$ is a constraint are as defined previously. Following the usual convention for reduction systems, we identify $\alpha$-renamable terms. That is, $\exists_x M$ and $\exists_x [y/x]M$ are regarded as the same term, provided that $y \notin \mathrm{fv}(M)$. Proposition 2.1 shows that this identification is consistent with our definition of a constraint system.

The standard operational model of **ccp** is given as a transition system over *configurations*. A configuration consists of a **ccp** agent and a constraint representing the current store. The transition system $T_D$ is specified with respect to a set of procedure declarations $D$. Fig. 1 gives the rules in the transition system. Constraints are added to the store (R1). A guarded choice is reduced non-deterministically by choosing a branch whose guard is enabled (R2). (R3) describes parallelism as interleaving. To describe locality (R4) the syntax of existentially quantified agents is extended by

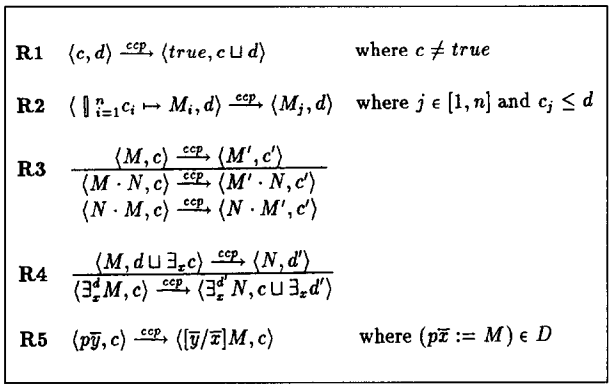


Fig. 1. The transition system $T_D$.

allowing agents of the form $\exists_x^d M$. This represents an agent in which $x$ is local to $M$ and $d$ is the "hidden" store that has been produced locally by $M$ on $x$. Initially, the local store is empty, that is, $\exists_x M = \exists_x^{true} M$. The execution of a procedure call is modelled by (R5). We write $\xrightarrow{ccp}$ for the reflexive and transitive closure of $\xrightarrow{ccp}$.

The standard observable behavior of a ccp agent is the set of possible constraint stores which can result when the agent is reduced to a normal form. A configuration $S$ is in *normal form* if it cannot be reduced further. Infinite reduction sequences are equated to the constraint *false*.

**Definition.** Let $P$ be the ccp program $D$; $M$. Then $P \Downarrow_{ccp} c$ if there is a normal form $\langle N, c\rangle$ such that $\langle M, true\rangle \xrightarrow{ccp} \langle N, c\rangle$ in the transition system $T_D$. $P$ *diverges*, written $P \Uparrow_{ccp}$ iff there is an infinite $T_D$-transition sequence starting with $\langle M, true\rangle$.

**Definition.** The set of *observations* of a program $P$, $Obs(\xrightarrow{ccp}, P)$ is

$$\{c \mid M \Downarrow_{ccp} c\} \cup \{false \mid M \Uparrow_{ccp}\}.$$

**Example 2.2.** The following declaration $D$ defines an agent *merge*, which non-deterministically merges its two input streams $x$ and $y$ into an output stream $z$. The constraint domain is equations over finite terms. We use [ ] to denote the empty stream, and $[u \mid v]$ to denote the stream with head $u$ and tail $v$:

$$\begin{aligned}
&merge(x,y,z) :=\\
&\quad \exists_{x'} \exists_u x = [u \mid x'] \mapsto \exists_{x'} \exists_u \exists_{z'} (x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x', y, z'))\\
&[] \ \exists_{y'} \exists_u y = [u \mid y'] \mapsto \exists_{y'} \exists_u \exists_{z'} (y = [u \mid y'] \cdot z = [u \mid z'] \cdot merge(x, y', z'))\\
&[] \ x = [\,] \mapsto z = y\\
&[] \ y = [\,] \mapsto z = x.
\end{aligned}$$

Let $P$ be the program $D;\ x = [a] \cdot merge(x, y, z) \cdot y = [b]$. A reduction sequence using left-most agent scheduling is

$$\langle x = [a] \cdot merge(x, y, z) \cdot y = [b], true\rangle$$

$$(R1) \xrightarrow{ccp} \langle merge(x, y, z) \cdot y = [b], x = [a]\rangle$$

$$(R5) \xrightarrow{ccp} \langle M \cdot y = [b], x = [a]\rangle$$

$$(R2) \xrightarrow{ccp} \langle \exists_{x'} \exists_u \exists_{z'} (x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x', y, z')) \cdot y = [b], x = [a]\rangle$$

$$(R1) \xrightarrow{ccp} \langle \exists_{x'}^{x'=[]} \exists_u^{u=a} \exists_{z'} (z = [u \mid z'] \cdot merge(x', y, z')) \cdot y = [b], x = [a]\rangle$$

$$(R1) \xrightarrow{ccp} \langle \exists_{x'}^{x'=[]} \exists_u^{u=a} \exists_{z'}^{z=[u \mid z']} (merge(x', y, z') \cdot y = [b]), x = [a]\rangle$$

$$(R5) \xrightarrow{ccp} \langle \exists_{x'}^{x'=[]} \exists_u^{u=a} \exists_{z'}^{z=[u \mid z']} (M' \cdot y = [b]), x = [a]\rangle$$

$$(R2) \xrightarrow{ccp} \langle \exists_{x'}^{x'=[]} \exists_u^{u=a} \exists_{z'}^{z=[u \mid z']} (y = z' \cdot y = [b]), x = [a]\rangle$$

$$(R1) \xrightarrow{ccp} \langle \exists_{x'}^{x'=[]} \exists_u^{u=a} \exists_{z'}^{z=[u \mid z'] \sqcup y=z'} (\cdot y = [b]), x = [a]\rangle$$

$$(R4) \xrightarrow{ccp} \langle true \cdot y = [b], x = [a] \sqcup z = [a \mid y]\rangle$$

$$(R1) \xrightarrow{ccp} \langle true \cdot true, y = [b] \sqcup x = [a] \sqcup z = [a, b]\rangle$$

where $M$ and $M'$ are appropriate renamings of the definition of $merge(x, y, z)$ and $merge(x', y, z')$, respectively. This reduction sequence gives the observable behavior

$$y = [b] \sqcup x = [a] \sqcup z = [a, b].$$

In fact, this is the only reduction sequence possible with a leftmost agent scheduling. With rightmost agent scheduling, however, the only observation is

$$y = [b] \sqcup x = [a] \sqcup z = [b, a].$$

Thus,

$$Obs(\xrightarrow{ccp}, P) \supseteq \{y = [b] \sqcup x = [a] \sqcup z = [b, a], y = [b] \sqcup x = [a] \sqcup z = [a, b]\}.$$

In fact, examination of the (large number of) other agent schedulings shows that these are the only observable behaviours. A more efficient way to show that these are the only observable behaviours will be discussed in the next section.

This example clearly shows the non-confluence of the standard operational semantics, as different agent schedulings give different results.

## 3. The concurrent constraint calculus

In this section, we develop a calculus for concurrent constraint programming which has the same observable behavior as the operational semantics defined in the last

section. The calculus is formulated as a reduction system modulo a set of structural congruences.

The calculus describes a slightly larger language than ccp, adding a blind choice operator $(+)$ and a failure operator $\mathbf{0}$, which is an identity for $(+)$. Informally, using $(+)$ one can collect all possible execution paths of an agent. We also admit a new form of guarded branch in an ask agent, written $\surd \rightarrow \mathbf{0}$, which stands for failure upon termination. Hence, a guard $g$ is now a constraint $c$ or the symbol $\surd$. Informally, once an alternative in a guarded choice is selected, the branch that corresponds to taking some other alternative is marked with a $\surd$-guard, which causes the branch to be discarded upon termination.

**Example 3.1.** To see the essential idea for obtaining confluence, consider the agent

$$A \stackrel{\text{def}}{=} d \mapsto M \;[]\; e \mapsto N,$$

run in a context where the store entails $d$. If the store does not also entail $e$ this should rewrite to $M$. On the other hand, if the store entails both $d$ and $e$, $A$ should rewrite to $M+N$. The problem is that the property "the store does not imply $e$" is not monotonic – in fact, it is anti-monotonic since the store increases monotonically during execution. Therefore, it is not possible to make a choice between the two reductions uniformly for all process schedulings. One solution to the problem is to consider each possible process scheduling individually, using an interpretation of parallel composition as interleaving. The resulting calculus is unsuitable for program analysis, however, due to the state space explosion incurred by the interleaving semantics.

In our calculus, $A$ reduces instead to

$$(M + (e \mapsto N \;[]\; \surd \mapsto 0) \stackrel{\text{def}}{=} B.$$

In effect, this defers the decision whether or not to drop the "$e \mapsto N$" branch until program termination. If further reductions determine that the store also entails $e$, this term could further reduce to

$$M + N + (\surd \mapsto 0 \;[]\; \surd \mapsto 0),$$

which is observationally equivalent to $M+N$. On the other hand, if the store never entails $e$, we end with agent $B$, which produces the same observations as $M$. We thus get a confluent calculus that is observationally equivalent to the transition system presented in the last section.

We now make these intuitions precise by defining a reduction system over an extended concurrent constraint language, called $\mathsf{ccp}_{+0}$. Terms in $\mathsf{ccp}_{+0}$ are produced by the grammar:

$$\text{(Agent)} \qquad M ::= c \mid R \mid p\overline{y} \mid M \cdot M \mid \exists_x M \mid M + M \mid \mathbf{0}$$

$$\text{(Choice)} \qquad R ::= R \;[]\; R \mid c \mapsto M \mid \surd \mapsto \mathbf{0}$$

The definitions of renaming and free variables carry over in the obvious way.

The operators have the natural precedence rules: $\exists_x$ binds strongest, followed by $(\cdot)$, followed by $([])$, followed by $(+)$ which binds weakest. Guard prefixes $g \mapsto$ extend as far to the right as possible.

The ccp calculus has a rich set of structural equivalences $(\equiv)$. If $M \equiv N$, then $M$ and $N$ are generally identified. If we want to avoid this identification, speaking only of the concrete term syntax, we will explicitly talk about pre-agents or pre-programs. Structural equivalence $(\equiv)$ is the least congruence that satisfies the laws below.

1. $(+)$ is associative and commutative, with identity $\mathbf{0}$:

$$(L + M) + N \equiv L + (M + N)$$
$$M + N \equiv N + M$$
$$M + \mathbf{0} \equiv M$$

2. $(\cdot)$ is associative and commutative, with identity *true* and zero $\mathbf{0}$:

$$(L \cdot M) \cdot N \equiv L \cdot (M \cdot N)$$
$$M \cdot N \equiv N \cdot M$$
$$M \cdot true \equiv M$$
$$M \cdot \mathbf{0} \equiv \mathbf{0}$$

3. $(\cdot)$ distributes through $(+)$:

$$M \cdot (N_1 + N_2) \equiv M \cdot N_1 + M \cdot N_2$$

4. $([])$ is associative and commutative:

$$(L \,[]\, M) \,[]\, N \equiv L \,[]\, (M \,[]\, N)$$
$$M \,[]\, N \equiv N \,[]\, M$$

5. Parallel composition of constraints equals least upper bound:

$$c \cdot c' \equiv c \sqcup c'$$

6. The following laws govern existential quantification:

$$\exists_x(M + N) \equiv \exists_x M + \exists_x N$$
$$M \cdot \exists_x N \equiv \exists_x(M \cdot N) \quad \text{if } x \notin \mathrm{fv}(M)$$
$$\exists_x M \equiv M \quad \text{if } x \notin \mathrm{fv}(M)$$
$$\exists_x M \equiv \exists_y [y/x]M \quad \text{if } y \notin \mathrm{fv}(M)$$
$$\exists_x \exists_y M \equiv \exists_y \cdot \exists_x M$$

Reduction $\rightarrow$ is a binary relation between agents that is parameterized by a procedure environment $D$ consisting of procedure declarations. We write $M \rightarrow_D N$ if $M$ reduces

to $N$ in one step in the procedure environment $D$. We sometimes leave out the $D$-suffix if the environment is clear from the context.

In essence there are two reduction rules, one for communication, and one for procedure unfolding. The rule for procedure unfolding is

$$p\overline{y} \xrightarrow{p}_D [\overline{y}/\overline{x}]M \quad (p\overline{x} := M \in D).$$

The rule for communication comes in two variants. The first variant handles the deterministic case, where no choice operator is present:

$$c \cdot (d \mapsto M) \xrightarrow{cc}_D c \cdot M \quad (d \leqslant c)$$

The second variant handles the case where the ask agent is part of a guarded choice:

$$c \cdot (d \mapsto M \,[]\, R) \xrightarrow{cc}_D c \cdot M + c \cdot (\surd \mapsto \mathbf{0} \,[]\, R) \quad (d \leqslant c)$$

The standard semantics of ccp captures the idea that once a guard in one of the guarded choice branches is enabled then that branch can be chosen and the other branches can be discarded. By contrast, our rule does not discard any branches. Instead, we also keep the original ask agent as a $(+)$-alternative, but with the taken branch replaced by the branch $(\surd \mapsto \mathbf{0})$.

Reduction can only occur in the top-level agents, it cannot occur inside the branches of a guarded choice. That is, our reduction relation, $\rightarrow$, is given by

$$\frac{M \xrightarrow{p \cup cc}_D M'}{\exists_{\overline{x}}\,(M \cdot N) + N' \rightarrow_D \exists_{\overline{x}}\,(M' \cdot N) + N'}.$$

We write $\twoheadrightarrow$ for the reflexive and transitive closure of $\rightarrow$.

We now define the set of possible observations of a ccp-term $M$. Since we express non-determinism by the $(+)$ operator, we might expect that each $(+)$-alternative in a reduct would contribute to the set of possible observations. However, we have to disregard those alternatives that contain a guard of the form $\surd \mapsto \mathbf{0}$ at top-level, since they represent untaken branches in a committed choice. Upon termination such alternatives are identified with failure, as is formalized below.

**Definition.** Let *terminal equivalence* $\approx$ be the least congruence that contains $\equiv$ and the equality

$$R \,[]\, \surd \mapsto \mathbf{0} \approx \mathbf{0}.$$

**Definition.** The *constraint part*, $Con(M)$, of a term $M$ is $\bigsqcup\{c \mid \exists N(M \equiv c \cdot N)\}$.

**Definition.** A term $M$ is in *normal form* if it cannot be reduced by $\rightarrow_D$.

**Definition.** Let $P$ be the $\mathsf{ccp}_{+0}$ program $D;M$. Then $P \Downarrow_{\mathsf{ccp}_{+0}} c$ if there is a normal form $N$ and a term $M'$ such that $M \twoheadrightarrow_D N + M', N \not\approx \mathbf{0}$ and $c = Con(N)$. $P$

*diverges*, written $P \Uparrow_{\mathrm{ccp}_{+0}}$ if there is an infinite $\rightarrow_D$-transition sequence starting with $M$.

The set of *observations* of a program $P$, $Obs(\rightarrow, P)$ is defined as in the ccp case.

$$Obs(\rightarrow, P) = \{c \mid M \Downarrow_{\mathrm{ccp}_{+0}} c\} \cup \{false \mid M \Uparrow_{\mathrm{ccp}_{+0}}\}.$$

Thus, the possible observations of a program $P$ are the constraint parts of all non-zero normal form alternatives of $P$. In addition, we add *false* to the observations of $P$ if there is a possibility that evaluation of $P$ does not terminate. We often abbreviate $Obs(\rightarrow, P)$ to $Obs(P)$.

As usual, we define *observational equivalence* ($\cong$) to be the largest congruence on terms and programs such that $P \cong Q$ implies $Obs(P) = Obs(Q)$, for all programs $P, Q$.

An equivalent, but more constructive definition of $\cong$ for terms is based on a *program context*, $C$, which is a program with a hole [ ] in it. Let $C[M]$ denote the term that results from filling out the hole in $C$ with $M$. Then $M \cong N$ iff for all program contexts $C$ such that $C[M]$ and $C[N]$ are well-formed programs,

$$Obs(C[M]) = Obs(C[N]).$$

**Proposition 3.2.** *The following are observational equivalences in* $\mathrm{ccp}_{+0}$:

$$M + M \cong M$$

$$M_1 + M_2 \cong true \mapsto M_1 \,[]\, true \mapsto M_2$$

$$R \,[]\, R \cong R$$

$$c \cdot (d \mapsto M \,[]\, R) \cong c \cdot R \qquad (c \sqcup d = false)$$

$$c \cdot (d \mapsto M \,[]\, \surd \mapsto \mathbf{0}) \cong c \cdot (d \mapsto M) \qquad (d \leqslant c)$$

Note that the second observational equivalence means that the explicit blind choice construct does not add to the expressiveness of ccp. The last observational equivalence holds because the guard is enabled.

**Example 3.3.** A reduction sequence in $\mathrm{ccp}_{+0}$ using leftmost agent scheduling from the program given in Example 2.2 is given in Fig. 2, where $M, M'$ and $M''$ are appropriate renamings of the definition of $merge(x, y, z), merge(x', y, z')$ and $merge(x, y', z')$ respectively and $R'$ and $R''$ are the remaining branches in the guarded choices in $M'$ and $M''$. This reduction sequence gives the observable behaviour

$$\{y = [b] \sqcup x = [a] \sqcup z = [b, a], y = [b] \sqcup x = [a] \sqcup z = [a, b]\}.$$

$$
\begin{array}{ll}
 & x = [a] \cdot merge(x,y,z) \cdot y = [b] \\
\xrightarrow{p} & y = [b] \sqcup x = [a] \cdot M \\
\xrightarrow{cc} & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'}\exists_{u}\exists_{z'}\ (x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x',y,z')) \\
 & +\exists_{y'}\exists_{u}\exists_{z'}\ (y = [u \mid y'] \cdot z = [u \mid z'] \cdot merge(x,y',z')) \\
 & +\surd \mapsto \mathbf{0} \parallel x = [] \mapsto z = y \parallel y = [] \mapsto z = x \\
 & ) \\
\cong & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'}\exists_{u}\exists_{z'}\ (x = [u \mid x'] \cdot z = [u \mid z'] \cdot merge(x',y,z')) \\
 & +\exists_{y'}\exists_{u}\exists_{z'}\ (y = [u \mid y'] \cdot z = [u \mid z'] \cdot merge(x,y',z')) \\
 & ) \\
\xrightarrow{p} & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'}\exists_{u}\exists_{z'}\ (x = [u \mid x'] \sqcup z = [u \mid z'] \cdot M') \\
 & +\exists_{y'}\exists_{u}\exists_{z'}\ (y = [u \mid y'] \sqcup z = [u \mid z'] \cdot M'') \\
 & ) \\
\xrightarrow{cc} & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'}\exists_{u}\exists_{z'}\ (x = [u \mid x'] \sqcup z = [u \mid z'] \cdot (z' = y + \surd \mapsto \mathbf{0} \parallel R')) \\
 & +\exists_{y'}\exists_{u}\exists_{z'}\ (y = [u \mid y'] \sqcup z = [u \mid z'] \cdot (z' = x + \surd \mapsto \mathbf{0} \parallel R'')) \\
 & ) \\
\cong & y = [b] \sqcup x = [a] \cdot \\
 & (\ \exists_{x'}\exists_{u}\exists_{z'}\ (x = [u \mid x'] \sqcup z = [u \mid z'] \sqcup z' = y) \\
 & +\exists_{y'}\exists_{u}\exists_{z'}\ (y = [u \mid y'] \sqcup z = [u \mid z'] \sqcup z' = x) \\
 & ) \\
\equiv & y = [b] \sqcup x = [a] \sqcup z = [a,b] + y = [b] \sqcup x = [a] \sqcup z = [b,a].
\end{array}
$$

Fig. 2. Example reduction in $\mathrm{ccp}_{+0}$.

This is exactly the observable behaviour with the ccp operational semantics, but is obtained with a single reduction scheduling.

## 4. Confluence

Example 2.2 demonstrated that with standard ccp reduction, $\xrightarrow{ccp}$, different agent schedulings can lead to different outcomes. In other words, the usual semantics of ccp languages is not confluent. In this section we show that $\rightarrow$, the reduction relation of $\mathrm{ccp}_{+0}$, is confluent. The confluence proof has to overcome the difficulty that agents do not form a free algebra (modulo $\alpha$-renaming), but are equivalence classes of pre-agents. Hence, standard techniques such as studied in [9] or [10] are not applicable.

Instead we adopt the following strategy: We define a *canonical form* $[\![M]\!]$ of a term $M$ (Section 4.1), together with a reduction relation on canonical forms (Section 4.2). We show that the canonical form mapping has an inverse, and that both it and its inverse commute with equivalences and multi-step reductions (Section 4.2). We then show that reduction on canonical forms is confluent, using standard techniques (Section 4.3). By the properties of the canonical form mapping, this gives us then confluence of the original $\mathrm{ccp}_{+0}$ calculus (Section 4.4). A similar technique has been used by Niehren and Smolka in their confluence proofs for the $\delta$ and $\rho$ calculi [15, 16].

### 4.1. Canonical forms

**Definition.** The syntax of *canonical forms* and their components is given below:

| | | |
|---|---|---|
| (Canonical form) | $X, Y ::= \{A_1, \ldots, A_k\}$ | $(k \geqslant 0)$ |
| (Alternative) | $A, B ::= \exists_{\bar{x}} c \bullet \mathscr{P} \bullet \mathscr{R}$ | $(\bar{x} \subseteq \mathrm{fv}(\mathscr{P}) \cup \mathrm{fv}(\mathscr{R}))$ |
| (Calls) | $\mathscr{P} ::= \{p_1 \bar{y}_1, \ldots, p_l \bar{y}_1\}$ | $(l \geqslant 0)$ |
| (Readers) | $\mathscr{R} ::= \{r_1, \ldots, r_m\}$ | $(m \geqslant 0)$ |
| (Reader) | $r ::= \{a_1, \ldots, a_n\}$ | $(n \geqslant 1)$ |
| (Guarded clause) | $a ::= c \mapsto X$ | |
| | $\mid \surd \mapsto \{\}$ | |

A *canonical form* $X$ is a multi-set of *alternatives*. Each alternative $A$ is of the form $\exists_{\bar{x}} c \bullet \mathscr{P} \bullet \mathscr{R}$; it consists of a set of existentially quantified variables $\bar{x}$, a constraint $c$, a multi-set $\mathscr{P}$ of procedure calls $p\bar{y}$, and a multi-set $\mathscr{R}$ of *readers* $r$. Each reader is in turn a non-empty multi-set of guarded clauses $c \mapsto X$ or $\surd \mapsto \{\}$. We require that for each alternative $\exists_{\bar{x}} c \bullet \mathscr{P} \bullet \mathscr{R}$ in a canonical form the set of existentially quantified variables $\bar{x}$ is contained in $\mathrm{fv}(\mathscr{P}) \cup \mathrm{fv}(\mathscr{R})$.

The set of free variables of canonical forms $X$ and their components is defined as follows:

$$
\begin{aligned}
\mathrm{fv}(\{A_1, \ldots, A_n\}) &= \mathrm{fv}(A_1) \cup \ldots \cup \mathrm{fv}(A_n) \\
\mathrm{fv}(\exists_{\bar{x}} c \bullet \mathscr{P} \bullet \mathscr{R}) &= (\mathrm{fv}(c) \cup \mathrm{fv}(\mathscr{P}) \cup \mathrm{fv}(\mathscr{R})) \backslash \bar{x} \\
\mathrm{fv}(\mathscr{P}) &= \bigcup \{\mathrm{fv}(p\bar{y}) \mid p\bar{y} \in \mathscr{P}\} \\
\mathrm{fv}(\mathscr{R}) &= \bigcup \{\mathrm{fv}(r) \mid r \in \mathscr{R}\} \\
\mathrm{fv}(r) &= \bigcup \{\mathrm{fv}(a) \mid a \in r\} \\
\mathrm{fv}(c \mapsto X) &= \mathrm{fv}(c) \cup \mathrm{fv}(X) \\
\mathrm{fv}(\surd \mapsto \{\}) &= \emptyset.
\end{aligned}
$$

We only consider canonical forms up to $\alpha$-renaming. That is, two alternatives $A \stackrel{\mathrm{def}}{=} \exists_{\bar{x}} c \bullet \mathscr{P} \bullet \mathscr{R}$ and $A' \stackrel{\mathrm{def}}{=} \exists_{\bar{x}'} c' \bullet \mathscr{P}' \bullet \mathscr{R}'$ are considered identical if $\bar{x} \cap \mathrm{fv}(A') = \bar{x}' \cap \mathrm{fv}(A) = \emptyset$ and there exists a renaming $\rho$ from $\bar{x}$ to $\bar{x}'$ such that $A' = \rho A$.

**Definition.** A *canonical form environment* is a set of procedure definitions $\{p\bar{x} := X\}$ that associate a procedure name $p$ and formal arguments $\bar{x}$ with a canonical form $X$. We use the letter $E$ for canonical form environments.

$$
\begin{array}{rcl}
[\![c]\!] & = & \{\exists_\emptyset\, c \bullet \emptyset \bullet \emptyset\} \\
[\![p\overline{y}]\!] & = & \{\exists_\emptyset\, true \bullet \{p\overline{y}\} \bullet \emptyset\} \\
[\![\exists_x M]\!] & = & \{\exists_x A \mid A \in [\![M]\!]\} \\
[\![M \cdot N]\!] & = & \{A \sqcup B \mid A \in [\![M]\!], B \in [\![N]\!]\} \\
[\![M + N]\!] & = & [\![M]\!] \cup [\![N]\!] \\
[\![0]\!] & = & \{\} \\
[\![c \mapsto M]\!] & = & \{\exists_\emptyset\, true \bullet \emptyset \bullet \{\{c \mapsto [\![M]\!]\}\}\} \\
[\![R_1 \parallel R_2]\!] & = & \{A \uplus B \mid A \in [\![R_1]\!], B \in [\![R_2]\!]\} \\
[\![D_1, D_2]\!] & = & [\![D_1]\!], [\![D_2]\!] \\
[\![p\overline{x} := M]\!] & = & p\overline{x} := [\![M]\!]
\end{array}
$$

$$
\begin{array}{rcll}
[\![\emptyset]\!]^{-1} & = & 0 & \\
[\![\{A_1, \ldots, A_n\}]\!]^{-1} & = & [\![A_1]\!]^{-1} + \ldots + [\![A_n]\!]^{-1} & (n \geq 1) \\
[\![\exists_{\overline{x}}\, c \bullet \{p_1\overline{y_1}, \ldots, p_j\overline{y_j}\} \bullet \{r_1, \ldots, r_k\}]\!]^{-1} & = & \exists_{\overline{x}}(c \cdot p_1\overline{y_1} \cdot \ldots \cdot p_j\overline{y_j} \cdot [\![r_1]\!]^{-1} \cdot \ldots \cdot [\![r_k]\!]^{-1}) & \\
[\![\{g_1 \mapsto X_1, \ldots, g_m \mapsto X_m\}]\!]^{-1} & = & g_1 \mapsto [\![X_1]\!]^{-1} \parallel \ldots \parallel g_m \mapsto [\![X_m]\!]^{-1} & \\
[\![E_1, E_2]\!]^{-1} & = & [\![E_1]\!]^{-1}, [\![E_2]\!]^{-1} & \\
[\![p\overline{x} := A]\!]^{-1} & = & p\overline{x} := [\![A]\!]^{-1} &
\end{array}
$$

Fig. 3. Mapping a term or procedure environment to its canonical form and back.

We now define some useful operations on alternatives of canonical forms. Let

$$A \stackrel{\text{def}}{=} \exists_{\overline{x}} c \bullet \mathscr{P} \bullet \mathscr{R}$$

$$A' \stackrel{\text{def}}{=} \exists_{\overline{x}'} c' \bullet \mathscr{P}' \bullet \mathscr{R}'$$

be two alternatives such that $\overline{x} \cap \overline{x}' = \overline{x} \cap \mathrm{fv}(A') = \overline{x}' \cap \mathrm{fv}(A) = \emptyset$. Then their *least upper bound* is given by

$$A \sqcup A' = \exists_{\overline{x}''}(c \sqcup c') \bullet (\mathscr{P} \cup \mathscr{P}') \bullet (\mathscr{R} \cup \mathscr{R}')$$

$$\text{where } \overline{x}'' = (\overline{x} \cup \overline{x}) \cap (\mathrm{fv}(\mathscr{P} \cup \mathscr{P}') \cup \mathrm{fv}(\mathscr{R} \cup \mathscr{R}'))$$

*Existential quantification* $\exists_x A$ of an alternative $A$ is defined as follows:

$$\exists_x(\exists_{\overline{y}} c \bullet \mathscr{P} \bullet \mathscr{R}) = \begin{cases} \exists_{\overline{y}}\ (\exists_x c) \bullet \mathscr{P} \bullet \mathscr{R} & \text{if } x \notin \mathrm{fv}(\mathscr{P}) \cup \mathrm{fv}(\mathscr{R}) \\ \exists_{\overline{y} \cup \{x\}} c \bullet \mathscr{P} \bullet \mathscr{R} & \text{if } x \in \mathrm{fv}(\mathscr{P}) \cup \mathrm{fv}(\mathscr{R}) \end{cases}$$

Another useful operation is the *merge*, $\uplus$, of two alternatives with a single reader each into an alternative where the guarded clauses in both readers are combined:

$$(\exists_{\overline{x}} c \bullet \mathscr{P} \bullet \{r_1\}) \uplus (\exists_{\overline{x}} c \bullet \mathscr{P} \bullet \{r_2\}) = \exists_{\overline{x}} c \bullet \mathscr{P} \bullet (r_1 \cup r_2).$$

### 4.2. Relationship between terms and canonical forms

We relate $\mathbf{ccp}_{+0}$ and canonical forms by means of a mapping, $[\![\cdot]\!]$, from a $\mathbf{ccp}_{+0}$ term to its canonical form. Fig. 3 defines this mapping together with its right inverse, $[\![\cdot]\!]^{-1}$.

**Lemma 4.1.** $[\![\cdot]\!]^{-1}$ *is well-defined. If* $X = Y$ *then* $[\![X]\!]^{-1} \equiv [\![Y]\!]^{-1}$.

**Proof.** Use $\alpha$-renaming and the associativity and commutativity laws to show that the names of bound variables and the order of existential quantifiers and subterms does not matter. □

**Lemma 4.2.** *For all terms* $M$, $[\![M]\!]^{-1} \equiv M$.

**Proof.** By a structural induction on the form of $M$. □

**Lemma 4.3.** *For all pre-terms* $M,N$, *we have* $M \equiv N$ *iff* $[M] = [N]$.

**Proof.** $\Rightarrow$: A tedious, but not very difficult induction on the derivation of $M \equiv N$.

$\Leftarrow$: Assume $[M] = [N]$. By Lemma 4.1, $[\![M]\!]^{-1} \equiv [\![N]\!]^{-1}$ . By Lemma 4.2, $M \equiv [\![M]\!]^{-1}$ and $N \equiv [\![N]\!]^{-1}$. Hence, $M \equiv N$. □

We now define a notion of reduction $\Rightarrow$ on canonical forms that simulates reduction $\rightarrow$ on $\mathsf{ccp}_{+0}$ terms. Analogous to $\rightarrow, \Rightarrow$ is parameterized by a normal form environment. There are three different ways a canonical form $X$ can reduce.

1. If $(p\bar{x} := Y) \in E$ and $X \equiv X' \cup \{\exists_{\bar{x}} c \bullet \{p\bar{y}\} \cup \mathscr{P} \bullet \mathscr{R}\}$ then

$$X \Rightarrow_E X' \cup \{(\exists_{\bar{x} \cap \mathrm{fv}(\mathscr{P},\mathscr{R})} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup A \mid A \in [\bar{y}/\bar{x}]Y\}.$$

2. If $d \leqslant c$ and $X \equiv X' \cup \{\exists_{\bar{x}} c \bullet \mathscr{P} \bullet \{\{d \mapsto Y\}\} \cup \mathscr{R}\}$ then

$$X \Rightarrow_E X' \cup \{(\exists_{\bar{x} \cap \mathrm{fv}(\mathscr{P},\mathscr{R})} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup A \mid A \in Y\}.$$

3. If $r \neq \emptyset$, $d \leqslant c$ and $X \equiv X' \cup \{\exists_{\bar{x}} c \bullet \mathscr{P} \bullet \{\{d \mapsto Y\} \cup r\} \cup \mathscr{R}\}$ then

$$X \Rightarrow_E X' \cup \{(\exists_{\bar{x} \cap \mathrm{fv}(\mathscr{P},\mathscr{R})} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup A \mid A \in Y\}$$
$$\cup \{(\exists_{\bar{x} \cap \mathrm{fv}(\mathscr{P},\mathscr{R},r)} c \bullet \mathscr{P} \bullet \{\{\surd \mapsto \{\}\} \cup r\} \cup \mathscr{R}\}\}.$$

We now show that multi-step $\Rightarrow$ reduction can simulate $\rightarrow$.

**Lemma 4.4.** *For all terms* $M,N$, *procedure environments* $D$, *if* $M \rightarrow_D N$, *then* $[M] \Rightarrow \Rightarrow_{[D]} [N]$.

**Proof.** A straightforward analysis of reduction rules establishes the result for top-level redexes. A structural induction on the context of a redex then establishes the result for all redexes. First assume that reduction $\rightarrow$ occurs at the root of the term $M$. We distinguish according to the form of reduction. If the redex is a procedure call, say, $p\bar{y}$ with $p\bar{x} := N$ in $D$, we have

$$\begin{aligned}
&[\![p\bar{y}]\!] \\
&= \{\exists_{\emptyset}\ \mathit{true} \bullet \{p\bar{y}\} \bullet \emptyset\} \\
&\Rightarrow \{(\exists_{\emptyset}\ \mathit{true} \bullet \emptyset \bullet \emptyset) \sqcup B \mid B \in [\![[\bar{y}/\bar{x}]N]\!]\} \\
&= [\![[\bar{y}/\bar{x}]N]\!].
\end{aligned}$$

If the redex is a communication, say $c \cdot (d \mapsto M \,[]\, R)$ where $d \leqslant c$, we distinguish according to whether $R$ is empty or non-empty. We do only the latter case here; the other case is similar, but simpler:

$$
\begin{aligned}
&[\![c \cdot (d \mapsto M \,[]\, R)]\!] \\
&= \{\exists_{\emptyset} c \bullet \emptyset \bullet \{\{d \mapsto [\![M]\!]\} \cup [\![R]\!]\}\} \\
&\Rightarrow \{(\exists_{\emptyset} c \bullet \emptyset \bullet \emptyset) \sqcup B \mid B \in [\![M]\!]\} \cup \{\exists_{\emptyset} c \bullet \emptyset \bullet \{\{\surd \mapsto \{\}\} \cup [\![R]\!]\}\} \\
&= [\![c \cdot M]\!] \cup [\![c \cdot (R \,[]\, \surd \mapsto 0)]\!] \\
&= [\![c \cdot M + c \cdot (R [] \surd \mapsto 0)]\!]
\end{aligned}
$$

Now assume that the redex occurs in a proper subterm of the term $M$. A structural induction on the context in which the redex occurs then shows the result. We do only one example case here; the others are similar. The case is as follows. Assume we have a term $M \cdot N$, and $N \to N'$. We have to show that $[\![M \cdot N]\!] \Rightarrow [\![M \cdot N']\!]$. Now,

$$[\![M \cdot N]\!] = \{A \sqcup B \mid A \in [\![M]\!], B \in [\![N]\!]\}.$$

By the induction hypothesis, $[\![N]\!] \twoheadrightarrow [\![N']\!]$. Since $\Rightarrow$-redexes are always single alternatives, there is for each $B \in [\![N]\!]$ a set $\{C_i \mid i \in I_B\}$ of alternatives such that $\{B\} \Rightarrow \{C_i \mid i \in I_B\}$ and $\bigcup\{C_i \mid i \in I_B, B \in [\![N]\!]\} = [\![N']\!]$. Since, furthermore, reduction is invariant under $\sqcup$ joins, i.e. $\{A\} \Rightarrow \{A'\}$ implies $\{A \sqcup B\} \Rightarrow \{A' \sqcup B\}$, we have that $\{A \sqcup B\} \Rightarrow \{A \sqcup C_i \mid i \in I_B\}$, for all $B$. Therefore,

$$
\begin{aligned}
&[\![M \cdot N]\!] \\
&= \{A \sqcup B \mid A \in [\![M]\!], B \in [\![N]\!]\} \\
&\twoheadrightarrow \bigcup\{\{A \sqcup C_i \mid i \in I_B\} \mid A \in [\![M]\!], B \in [\![N]\!]\} \\
&= \{A \sqcup C_i \mid A \in [\![M]\!], B \in [\![N]\!], i \in I_B\} \\
&= \{A \sqcup B' \mid A \in [\![M]\!], B' \in [\![N']\!]\} \\
&= [\![M \cdot N']\!]. \qquad \square
\end{aligned}
$$

The reverse of Lemma 4.4 also holds.

**Lemma 4.5.** *For all canonical forms $X, Y$, canonical form environments $E$, if $X \Rightarrow_E Y$, then $[\![X]\!]^{-1} \to_{[\![E]\!]^{-1}} [\![Y]\!]^{-1}$.*

**Proof.** A straightforward case analysis on the kind of reduction $\Rightarrow$. $\square$

### 4.3. Confluence of canonical form reduction

We now establish that reduction $\Rightarrow$ is confluent. We do this by first considering reductions for procedure unfoldings and communications independently of each other.

**Definition.** Let $\overset{p}{\Rightarrow}$ be the reduction relation generated by the first rule (the unfolding rule) in the definition of $\Rightarrow$. Let $\overset{cc}{\Rightarrow}$ be the reduction relation generated by the second and third rule (the communication rules) in the definition of $\Rightarrow$.

**Lemma 4.6.** *$\overset{p}{\Rightarrow}$ is Church–Rosser: If $X \overset{p}{\Rightarrow}_E X_1$ and $X \overset{p}{\Rightarrow}_E X_2$ then there is a canonical form $X_3$ s.t. $X_1 \overset{p}{\Rightarrow}_E X_3$ and $X_2 \overset{p}{\Rightarrow}_E X_3$.*

**Proof.** This is essentially a first-order restriction of the Church–Rosser theorem in $\lambda$-calculus. It can be shown by adapting Plotkin's confluence proof for $\lambda_V$ [18], showing confluence of parallel reductions as an intermediate step. □

**Lemma 4.7.** *$\overset{cc}{\Rightarrow}$ is weakly Church–Rosser: If $X \overset{cc}{\Rightarrow} X_1$ and $X \overset{cc}{\Rightarrow} X_2$ then there is a canonical form $X_3$ s.t. $X_1 \overset{cc}{\Rightarrow} X_3$ and $X_2 \overset{cc}{\Rightarrow} X_3$.*

**Proof.** A standard analysis of critical pairs. We do only one example case here.

Let $X = \{\exists_{\overline{x}} c \bullet \mathscr{P} \bullet \{\{d_1 \mapsto M_1, d_2 \mapsto M_2\} \cup r\} \cup \mathscr{R}\} \cup X'$ be a canonical form such that $d_1 \leqslant c$ and $d_2 \leqslant c$. Let

$$\begin{aligned}
\overline{y} &= \overline{x} \cap \mathrm{fv}(\mathscr{P}, \mathscr{R}) \\
\overline{y}_0 &= \overline{x} \cap \mathrm{fv}(\mathscr{P}, \mathscr{R}, r) \\
\overline{y}_1 &= \overline{y} \cap \mathrm{fv}(\mathscr{P}, \mathscr{R}, r, d_1, M_1) \\
\overline{y}_2 &= \overline{y} \cap \mathrm{fv}(\mathscr{P}, \mathscr{R}, r, d_2, M_2)
\end{aligned}$$

Then

$$X \Rightarrow \{(\exists_{\overline{y}} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup B \mid B \in M_1\}$$

$$\cup \{\exists_{\overline{y}_2} c \bullet \mathscr{P} \bullet \mathscr{R} \cup \{d_2 \mapsto M_2, \surd \mapsto \{\}\} \cup r\} \cup X' \overset{\text{def}}{=} X_1$$

and also

$$X \Rightarrow \{(\exists_{\overline{y}} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup B \mid B \in M_2\}$$

$$\cup \{\exists_{\overline{y}_1} c \bullet \mathscr{P} \bullet \mathscr{R} \cup \{d_1 \mapsto M_1, \surd \mapsto \{\}\} \cup r\} \cup X' \overset{\text{def}}{=} X_2$$

But then for $i = 1, 2$:

$$X_i \Rightarrow \{(\exists_{\overline{y}} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup B \mid B \in M_1\}$$

$$\cup \{(\exists_{\overline{y}} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup C \mid C \in M_2\}$$

$$\cup\{\exists_{\overline{y}_0} c \bullet \mathscr{P} \bullet \mathscr{R} \cup \{\surd \mapsto \{\}, \surd \mapsto \{\}\} \cup r\}$$

$$\cup\{\exists_{\overline{y}_0} c \bullet \mathscr{P} \bullet \mathscr{R} \cup \{\surd \mapsto \{\}, \surd \mapsto \{\}\} \cup r\}.$$

The other cases are similarly straightforward. □

**Lemma 4.8.** $\overset{cc}{\Rightarrow}$ *is Noetherian: every sequence of* $\overset{cc}{\Rightarrow}$ *reductions has finite length.*

**Proof.** Define a norm $\|\cdot\|$ that assigns non-negative integers to canonical forms and alternatives as follows:

$$\|\{A_i\}_{i\in I}\| = \sum_{i\in I} \|A_i\|$$

$$\|\exists_{\overline{x}} c \bullet \mathscr{P} \bullet \mathscr{R}\| = \prod_{r\in\mathscr{R}} \left(1 + \sum_{(d\mapsto Y)\in r} \|Y\|\right)$$

An inspection of the reduction rules for $\overset{cc}{\Rightarrow}$ shows that $\|X\| > \|X'\|$ whenever $X \overset{cc}{\Rightarrow} X'$. □

**Lemma 4.9.** $\overset{cc}{\Rightarrow}$ *is Church–Rosser.*

**Proof.** A direct consequence of Lemmas 4.7 and 4.8, using Newman's lemma [1, Proposition 3.1.25]. □

**Lemma 4.10.** $\overset{p}{\Rightarrow}$ *and* $\overset{cc}{\Rightarrow}$ *commute: For all canonical forms* $X, X_1, X_2$ *such that* $X \overset{p}{\Rightarrow} X_1$ *and* $X \overset{cc}{\Rightarrow} X_2$ *there is a canonical form* $X_3$ *such that* $X_1 \overset{cc}{\Rightarrow} X_3$ *and* $X_2 \overset{p}{\Rightarrow} X_3$.

**Proof.** Assume $X \overset{p}{\Rightarrow} X_1$ and $X \overset{cc}{\Rightarrow} X_2$. Then simply repeat all $\overset{cc}{\Rightarrow}$ steps in the corresponding $\overset{p}{\Rightarrow}$ residuals in $X_1$ of the alternatives in $X$. Likewise, repeat all $\overset{p}{\Rightarrow}$ steps in the corresponding $\overset{cc}{\Rightarrow}$ residuals in $X_2$ of the alternatives in $X$. An inspection of the reduction rules for $\overset{p}{\Rightarrow}$ and $\overset{cc}{\Rightarrow}$ shows that this yields the same canonical form. □

**Lemma 4.11.** $\Rightarrow$ *is Church–Rosser.*

**Proof.** By Lemmas 4.6 and 4.9, $\overset{p}{\Rightarrow}$ and $\overset{cc}{\Rightarrow}$ are both Church–Rosser and by Lemma 4.10 they commute with each other. By the lemma of Hindley and Rosen [1, Proposition 3.3.5], it follows that $\Rightarrow \equiv \overset{p}{\Rightarrow} \cup \overset{cc}{\Rightarrow}$ is Church–Rosser. □

### *4.4. Confluence of* $\mathsf{ccp}_{+0}$*-reduction*

We are finally in a position to show confluence for the original notion of reduction $\rightarrow$ on $\mathsf{ccp}_{+0}$ terms.

**Theorem 4.12.** $\rightarrow$ *is Church–Rosser. For all terms* $M, M_1, M_2$, *environments* $D$, *if* $M \twoheadrightarrow_D M_1$ *and* $M \twoheadrightarrow_D M_2$ *then there is a term* $M_3$ *s.t.* $M_1 \twoheadrightarrow_D M_3$ *and* $M_2 \twoheadrightarrow_D M_3$.

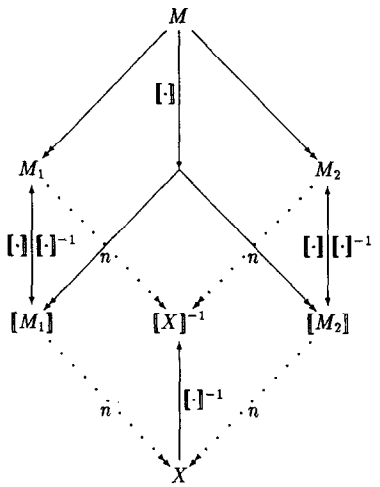


Fig. 4. Strategy of the CR proof.

**Proof.** The proof strategy is depicted in Fig. 4. Assume that $M \twoheadrightarrow_D M_1$ and $M \twoheadrightarrow_D M_2$. By an induction on the length of the two reduction sequences from $M$ to $M_1$ and $M_2$, using Lemmas 4.3 and 4.4 at each step, we have that $[\![M]\!] \twoheadrightarrow_{[\![D]\!]} [\![M_1]\!]$ and $[\![M]\!] \twoheadrightarrow_{[\![D]\!]} [\![M_2]\!]$. Since by Lemma 4.11 $\Rightarrow$ is confluent, this implies the existence of a canonical form $X$ such that $[\![M_1]\!] \twoheadrightarrow_{[\![D]\!]} X$ and $[\![M_2]\!] \twoheadrightarrow_{[\![D]\!]} X$. By Lemma 4.2, $[\![\cdot]\!]^{-1}$ is an inverse of $[\![\cdot]\!], [\![[\![M_i]\!]]\!]^{-1} \equiv M_i$ for $i = 1,2$. Then by induction on the length of the two reduction sequences from $[\![M_1]\!]$ and $[\![M_2]\!]$ to $X$, using Lemma 4.5 at each step, we have that $M_i = [\![[\![M_i]\!]]\!]^{-1} \twoheadrightarrow [\![X]\!]^{-1}$, $(i = 1,2)$. This implies the proposition with $M_3 = [\![X]\!]^{-1}$. □

## 5. Relationship to ccp

In this section we show that the observational behaviour of our calculus is identical to the observational behaviour of **ccp** in its standard transition system semantics. To do this we extend $[\![\cdot]\!]$ so that it maps a **ccp** configuration to a subset of the canonical forms given in the previous section, together with a reduction relation $\overset{ccp}{\Longrightarrow}$ on this canonical form and a notion of observables. We show that for a given program $\overset{ccp}{\longrightarrow}, \overset{ccp}{\Longrightarrow}$, $\Rightarrow$ and $\rightarrow$ all give rise to the same observations.

In order to extend $[\![\cdot]\!]$, we first give a mapping $pa(\ )$ from **ccp** agents in a configuration to a $\mathbf{ccp}_{+0}$ pre-agent. This is needed because **ccp** agents in a configuration may have hidden stores which are not allowed in pre-agents:

$$
\begin{aligned}
pa(c) &= c \\
pa(p\overline{y}) &= p\overline{y} \\
pa(\exists_x^d M) &= \exists_x(d \cdot pa(M)) \\
pa(M \cdot N) &= pa(M) \cdot pa(N) \\
pa(g \mapsto M) &= g \mapsto M.
\end{aligned}
$$

Note that terms in the range of $pa$ never contain $\mathbf{0}$, $+$ or $\surd$. The *canonical form* of a ccp configuration $\langle A, c\rangle$ is given by

$$[\![\langle A, c\rangle]\!] = [\![pa(A) \cdot c]\!].$$

As ccp agents and programs do not contain blind choice, the canonical form of a ccp configuration will always consist of a single alternative. Because there is no need to distribute blind choice over the parallel operator, there is a bijection between the readers and the procedure calls in the ccp configuration and the canonical form. We will make use of this correspondence in the proofs below.

We now define a notion of reduction $\overset{ccp}{\Longrightarrow}$ on the canonical form of a ccp agent that simulates reduction $\overset{ccp}{\longrightarrow}$ on ccp configurations. Like $\Rightarrow$, $\overset{ccp}{\Longrightarrow}$ is parameterized by an environment $E$ of definitions, i.e., associations between ccp procedure names with formal arguments and the canonical form of their definition. There are two different ways a ccp canonical form $X$ can reduce:

1. If $(p\bar{x} := Y) \in E$ and $\{A\} = [\bar{y}/\bar{x}]Y$ then

$$\{\exists_{\bar{x}} c \bullet \{p\bar{y}\} \cup \mathscr{P} \bullet \mathscr{R}\} \overset{ccp}{\Longrightarrow}_E \{(\exists_{\bar{x} \cap \mathrm{fv}(\mathscr{P},\mathscr{R})} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup A\}.$$

2. If $d \leqslant c$ then and $\{A\} = Y$

$$\{\exists_{\bar{x}} c \bullet \mathscr{P} \cdot \{\{d \mapsto Y\} \cup r\} \cup \mathscr{R}\} \overset{ccp}{\Longrightarrow}_E \{(\exists_{\bar{x}} c \bullet \mathscr{P} \bullet \mathscr{R}) \sqcup A\}.$$

**Definition.** A canonical form is in *normal form* if it cannot be reduced. $Con(A)$ is the constraint component of $A$. We write $\overset{ccp}{\Longrightarrow}$ for the reflexive and transitive closure of $\overset{ccp}{\Longrightarrow}$.

Analogous to the cases for $\rightarrow$ reductions and $\overset{ccp}{\longrightarrow}$ transitions, we now define two notions of observables for canonical form reductions.

**Definition.** Let the notion of reduction $\hookrightarrow$ be one of $\Rightarrow$, $\overset{ccp}{\Longrightarrow}$. Let $P$ be the ccp program $D; M$. Then the set of possible observations of $P$ wrt $\hookrightarrow$ is given by

$$Obs(\hookrightarrow, P) = \bigcup\{Obs([\![A]\!]^{-1}) \mid P \subset\!\!\!\rightarrow \{A\} \cup X \text{ and } \{A\} \text{ is in } \hookrightarrow\text{-normal form}\}.$$

The following two lemmas are shown by an analysis of $\overset{ccp}{\longrightarrow}$ transitions and $\overset{ccp}{\Longrightarrow}$ reductions.

**Lemma 5.1.** *If* $S \overset{ccp}{\longrightarrow} S'$ *in the transition system* $T_D$, *then either* $[\![S]\!] = [\![S']\!]$ *or* $[\![S]\!] \overset{ccp}{\Longrightarrow}_{[\![D]\!]} [\![S']\!]$.

**Proof.** Consider the atomic subagent $A$ in $S$ which has been reduced. There are three cases.

- If $A$ is a constraint then $[\![S]\!] = [\![S']\!]$.
- If $A$ is a procedure call then we can choose the corresponding procedure call in $[\![S]\!]$ and reduce this call to give $X$. It is straightforward to verify that $X = [\![S']\!]$.

- If $A$ is a choice $[]_{i=1}^{n} c_i \mapsto A_i$ and the $i$th guarded branch is chosen, then we can choose the corresponding reader in $[\![S]\!]$ and reduce this to give $X$. The difficulty is to prove that the corresponding reader is enabled in $[\![S]\!]$. This follows because we can rename the variables in the hidden stores so that they do not interfere with each other, and then move the existential quantifiers to the start of the constraint lub. It is straightforward to verify that $X = [\![S']\!]$. □

**Lemma 5.2.** *Let $S$ be a* ccp *configuration and $D$ be a set of* ccp *definitions. If $[\![S]\!] \xrightarrow{ccp}_{[\![D]\!]} X$ then there is a configuration $S'$ such that $X = [\![S']\!]$ and $S \xrightarrow{ccp}\!\!\!\!\rightarrow S'$ in the transition system $T_D$.*

**Proof.** Let $S''$ be the configuration obtained from $S$ by reducing all constraints in $S$ which are not in a choice. Then $[\![S'']\!] = [\![S]\!]$ and $S \xrightarrow{ccp}\!\!\!\!\rightarrow S''$. Now consider the subagent $A$ in $[\![S]\!]$ which has been reduced. There are two cases.

- If $A$ is a procedure call then we can choose the corresponding procedure call in $S''$ and reduce this call to give $S'$. It is straightforward to verify that $X = [\![S']\!]$.
- If $A$ is a reader and the $i$th guarded clause is chosen, then we can choose the corresponding choice and branch in $S''$ and reduce this to give $S'$. The difficulty is to prove that the corresponding guard is enabled in $S''$. Again this follows because we can rename the variables in the hidden stores so that they do not interfere with each other, and then move the existential quantifiers to the start of the constraint lub. It is straightforward to verify that $X = [\![S']\!]$. □

**Lemma 5.3.** *Let $S$ be a* ccp *configuration. If $S$ is in normal form, then $[\![S]\!]$ is in normal form and $Con(S) = Con([\![S]\!])$. Furthermore, if $[\![S]\!]$ is in normal form, then there is a $S'$ such that $S \xrightarrow{ccp}\!\!\!\!\rightarrow S'$, $S'$ is in normal form and $[\![S']\!] = [\![S]\!]$.*

**Proof.** It follows from Lemma 5.2 that if $[\![S]\!]$ is not in normal form, that is it can be reduced, then $S$ can also be reduced and so is not in normal form. Thus, $S$ is in normal form, then $[\![S]\!]$ is in normal form. It is straightforward to verify that $Con(S) = Con([\![S]\!])$.

If $[\![S]\!]$ is in normal form, then from Lemma 5.1, then the only reductions that can occur in $S$ are reductions of contraints. Let $S'$ be the configuration obtained from $S$ by reducing all constraints. From the definition of normal form, $[\![S]\!] = [\![S']\!]$. Hence, as $[\![S]\!]$ is in normal form, no procedure call or reader can be reduced $S'$ and so $S'$ is in normal form. □

Thus:

**Lemma 5.4.** *For any* ccp *program $P$, $Obs(\xrightarrow{ccp}, P) = Obs(\Longrightarrow^{ccp}, P)$.*

**Proof.** We first prove that $Obs(\xrightarrow{ccp}, P) \supseteq Obs(\Longrightarrow^{ccp}, P)$. If false $\in Obs(\Longrightarrow^{ccp}, P)$ and $P$ has an infinite $\Longrightarrow^{ccp}$-reduction sequence, it follows from Lemma 5.2 that $P$ will have an infinite $\xrightarrow{ccp}$-reduction sequence. Thus *false* $\in Obs(\xrightarrow{ccp}, P)$. Otherwise there

is a reduction sequence with $\stackrel{ccp}{\Longrightarrow}$ ending in a normal form $X$ with $c = Con(X)$. From Lemma 5.2 and the definition $[\![ \cdot ]\!]$ on configurations and Lemma 5.3 there is a corresponding reduction sequence with $\stackrel{ccp}{\longrightarrow}$ ending in a normal form configuration $\langle N, c\rangle$. Thus $c \in Obs(\stackrel{ccp}{\longrightarrow}, P)$. The other direction, $Obs(\stackrel{ccp}{\Longrightarrow}, P) \subseteq Obs(\stackrel{ccp}{\longrightarrow}, P)$, follows by an analogous argument. □

We also have that:

**Lemma 5.5.** *For any* ccp *program* $P, Obs(\stackrel{ccp}{\Longrightarrow}, P) = Obs(\Rightarrow, P)$.

**Proof.** We first prove that $Obs(\stackrel{ccp}{\Longrightarrow}, P) \subseteq Obs(\Rightarrow, P)$. Consider the reduction $\{X_1\} \stackrel{ccp}{\Longrightarrow} \{X_2\} \stackrel{ccp}{\Longrightarrow} \cdots \stackrel{ccp}{\Longrightarrow} \{X_n\}$. By performing essentially the same reductions it is straightforward to construct a reduction $Y_1 \Rightarrow Y_2 \Rightarrow \cdots \Rightarrow Y_n$ such that for each $Y_i$, $X_i \in Y_i$. It follows that $Obs(\stackrel{ccp}{\Longrightarrow}, P) \subseteq (Obs(\Rightarrow, P)$.

Proving that $Obs(\stackrel{ccp}{\Longrightarrow}, P) \supseteq Obs(\Rightarrow, P)$ is slightly more difficult. Consider the reduction sequence $Y_1 \Rightarrow Y_2 \Rightarrow \cdots \Rightarrow Y_n$ where $Y_1 = \{A_1\}$ and $A_n \in Y_n$ is in normal form. We first inductively construct a sequence $X_1, X_2, \ldots, X_n$ where each $X_i$ consists of just the alternative in $Y_i$ which is the "ancestor" of $X_{i+1}$ and $X_n$ is $\{A_n\}$. Now readers in an $X_i$ may have guarded branches containing an invalidator; we modify these to obtain a new sequence $X_1', X_2', \ldots, X_n'$ as follows. Consider a reader $R$ occurring in an $X_i$ without any invalidated branches. In subsequent alternatives $X_{i+1}, \ldots$ the reader may be reduced so that a branch is replaced with an invalidator. We replace all of these descendant readers of $R$ by $R$ itself. Note that every reader in $X_1, X_2, \ldots, X_n$ must have an ancestor which does not contain any invalidated branches. Thus, our new sequence $X_1', X_2', \ldots, X_n'$ will not contain any guarded branches with an invalidator. It is straightforward to verify that for each $X_i'$, either $X_i' = X_{i+1}'$ or else $\{X_i'\} \stackrel{ccp}{\Longrightarrow} \{X_{i+1}'\}$. Also, $X_n' = X_n$. Thus, $Obs(\stackrel{ccp}{\Longrightarrow}, P) \supseteq Obs(\Rightarrow, P)$. □

From Lemmas 4.4 and 4.5:

**Lemma 5.6.** *For any program* $P, Obs(\Rightarrow, P) = Obs(P)$.

The main result of this section follows from Lemmas 5.4, 5.5 and 5.6 – the confluent calculus is observationally equivalent to the operational semantics of ccp.

**Theorem 5.7.** *For any* ccp *program* $P, Obs(P) = Obs(\stackrel{ccp}{\longrightarrow}, P)$.

## 6. Application to program analysis

One application of our confluent semantics is to the static analysis of ccp programs. Lack of confluence in the usual operational semantics and denotational semantics of ccp languages means that program analysis cannot be directly based on these semantics, as the cost of considering all process schedulings in an analysis is prohibitive. There have

been two main approaches to overcome this difficulty. The first is to use a fixed process scheduling, but then to "re-execute" the program until a fixpoint is reached. This was suggested in [5] for concurrent logic programs and extended in [6] to ccp. This may be expensive and is inherently imprecise because re-execution confuses the behaviour of different branches. The second approach is to give a non-standard operational semantics for ccp which is confluent but which approximates the usual ccp operational semantics by allowing more reductions. This was suggested in [2, 3] for concurrent logic programs and couched in [21, 7] in the slightly different context of ccp as a transformation from a program written in full ccp to an approximating program written in a subset of ccp for which the usual operational semantics is confluent. Codognet and Codognet [4] use a similar idea as the basis for program analysis. They introduce a new type of guarded choice which has a confluent semantics.

Our calculus provides an alternative semantic basis for program analysis. Because the calculus is Church–Rosser it has all of the advantages of the approximate confluent semantics or program transformation as the basis for program analysis. It has the additional advantages that there is no need for a complex and artificial approximate semantics and that it is inherently more precise because programs have exactly the same observable behaviour as in the usual operational semantics and the calculus does not introduce extra reductions.

For example, consider the ccp agent

$$x = a \cdot choose(x, y, z) \cdot (z = a \mapsto true)$$

with the following ccp definition:

$$choose(x, y, z) := x = a \mapsto z = x \;[]\; y = a \mapsto true.$$

The approximate confluent semantics of [3][1] will introduce the reduction sequence

$$\begin{aligned}
&\langle x = a \cdot choose(x, y, z) \cdot (z = a \mapsto true), true\rangle \\
\mapsto\ &\langle choose(x, y, z) \cdot (z = a \mapsto true), x = a\rangle \\
\to\ &\langle (x = a \mapsto z = x \;[]\; y = a \mapsto true) \cdot (z = a \mapsto true), x = a\rangle \\
\to\ &\langle true \cdot (z = a \mapsto true), x = a \sqcup y = a\rangle
\end{aligned}$$

as in the approximate semantics, once one branch in a guard is enabled, all branches are assumed to be enabled. This extra reduction sequence ends in an agent which is "suspended" in the sense that it consists of blocked readers. The other semantics given in [21, 7] will also introduce an equivalent reduction sequence. This is unfortunate as it means that no analysis based on the approximate confluent semantics or transformed program approach can ever prove that this agent is suspension free which is currently the most important application of ccp analysis. However, an analysis based on our calculus can (correctly) show that this agent can never lead to suspension.

[1] Note that the analysis given in [3] is for concurrent constraint logic programs and so allows "tell" constraints in guards. To use the analysis we treat guards as having the *true* tell constraint.

## 7. Conclusion

We have given a calculus for a class of languages, $\mathsf{ccp}_{+0}$, which generalizes concurrent constraint programs (ccp). However, unlike the usual operational semantics for ccp, the calculus is confluent in the sense that different process schedulings give rise to exactly the same set of possible outcomes. This disproves the folklore that it is impossible to give a confluent semantics for languages with non-deterministic guarded choice.

The calculus has application to static analysis of ccp programs. As the calculus is confluent, it provides a good basis on which to develop analyses. Confluence means that not all process schedulings need to be considered in an analysis, allowing for efficiency, and that an analysis can choose a process scheduling which gives better information, allowing for accuracy.

## Acknowledgements

We thank the referees of CP'95 and TCS for their detailed comments.

## References

[1] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, Vol. 103 (North-Holland, Amsterdam, revised ed., 1984).

[2] M. Codish, M. Falaschi and K. Marriott, Suspension analyses for concurrent logic programs, *ACM Trans. Programming Languages Systems* **16** (1994) 649–686.

[3] M. Codish, M. Falaschi, K. Marriott and W. Winsborough, Efficient analysis of concurrent constraint logic programs, in: *Proc. 20th Internat. Coll. on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 700 (Springer, Berlin, 1993) 633–644.

[4] C. Codognet and P. Codognet, Guarded constructive disjunction: angel or demon? in: *Proc Internat. Conf. on Principles and practice of Constraint Programming, CP'95*, Marseille, France, Lecture Notes in Computer Science, Vol. 976 (Springer, Berlin, 1995) 345–361.

[5] C. Codognet, P. Codognet and M. Corsini, Abstract interpretation for concurrent logic languages, in: *Proc. North American Conf. on Logic Programming* (1990) 215–232.

[6] M. Falaschi, M. Gabbrielli, K. Marriott and C. Palamidessi, Compositional analysis for concurrent constraint programming, in: *Proc. 8th IEEE Symp. on Logic in Computer Science* (1993) 210–221.

[7] M. Falaschi, M. Gabbrielli, K. Marriott and C. Palamidessi, Confluence and concurrent constraint programming, in: V.S. Alagar and M. Nivat, eds., *Proc. 4th Internat. Conf. on Algebraic Methodology and Software Technology (AMAST'95)*, Montreal, Canada, Lecture Notes in Computer Science, Vol. 936 (Springer, Berlin, 1995) 531–545.

[8] L. Henkin, J.D. Monk and A. Tarski, *Cylindric Algebras*, Studies in Logic and the Foundations of Mathematics, Vol. 64 (North-Holland, Amsterdam, 1971).

[9] G. Huet, Confluent reductions: Abstract properties and applications to term rewriting systems, *J. ACM* **27** (1980) 797–821.

[10] Jan Willem lop, Combinatory reduction systems. Ph.D. Thesis, Mathematisch Centrum, Kruislaan 413, 1098 SJ Amsterdam, 1980. Mathematical Centre Tracts, Vol. 127.

[11] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, 2nd ed., 1987).

[12] M. Maher, Logic semantics for a class of committed-choice programs, in: *Proc. 4th Internat. Conf. on Logic Programming* (1987) 858–876.

[13] K. Marriott, M. Falaschi, M. Gabbrielli and C. Palamidessi, A simple semantics for logic programming languages with delay, in: R. Kotagiri, ed., *Proc. 18th Australian Computer Science Conf., Australian Computer Science Comm.* **17** (1995) 356–363.

[14] U. Montanari, F. Rossi and V. Saraswat, CC programs with both in- and non-determinism: A concurrent semantics, in: *Proc. 2nd Internat. Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, Lecture Notes in Computer Science, Vol. 874 (Springer, Berlin, 1994) 151–161.

[15] J. Niehren, Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen. Ph.D. Thesis, Universität des Saarlandes, 1994.

[16] J. Niehren and G. Smolka, A confluent relational calculus for higher-order programming with constraints, in: J.-P. Jouannaud, ed., *Proc. 1st Internat. Conf. on Constraints in Computational Logics*, München, Germany, Lecture Notes in Computer Science, Vol. 845 (Springer, Berlin, 1994) 89–104.

[17] S.-O. Nyström, *Oracles and confluence*–A fixpoint semantics for concurrent constraint programming, Tech. Report 115. Computer Science Department, Uppsala University. 1995.

[18] G.D. Plotkin. Call-by-name, call-by-value, and the $\lambda$-calculus. *Theoret. Comput. Sci.* **1** (1975) 125 159.

[19] V.A. Saraswat and M. Rinard, Concurrent constraint programming, in: *Proc. 17th Ann. ACM Symp. on Principles of Programming Languages*, San Francisco, CA (1990) 232–245.

[20] V. Saraswat, M. Rinard and P. Panangaden, The semantic foundations of concurrent constraint programming, in: *Conf. Record of the 18th Ann. ACM Symp. on Principles of Programming Languages*, Orlando, Florida (ACM Press, New York, 1991) 333–352.

[21] E. Zaffanella, G. Levi and R. Giacobazzi, Abstracting synchronization in concurrent constraint programming, in: *Proc. 5th Internat. Symp. on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, Vol. 844 (Springer, Berlin, 1994).